Principles of Performance Tuning

Klaus Marquardt, Dorothea-Erxleben-Straße 78, Germany

Email: pattern@kmarquardt.de

Every software engineer has used techniques to increase the execution speed of the system under development. Some of these focus on design and implementation, like using a more efficient algorithm. Others focus on correct distribution of tasks among different system parts, on feature avoidance, or on appropriate project and process measures.

Engineers working in different areas are used to apply specific techniques, but most of these different and occasionally unique techniques follow the same basic principles. These common principles and a number of their applications are presented here.

Introduction

Few, if any, projects did not suffer from performance problems at some time. Sometimes the hardware is way too slow, all the development tools and processes are inefficient, and in the end the software itself lacks responsiveness and reacts late to processing requests.

Performance of software systems has a lot of aspects, such as:

Project Solutions. Increasing responsiveness is often dealt against decreasing another valuable resource. As responsiveness is a quality, the most significant resources in a software project to trade off are scope, cost, time, and other quality aspects.
In many systems, namely embedded systems, software is an integral part, but only one part of the system. Here trade-offs against hardware resources are a common approach.

Design Solutions. The most frequent, and for programmers most natural approach for creating responsiveness is to increase their effort. This effort can be spend in sophisticated algorithmic solutions, or in strategies that apply during runtime.

Process Solutions. Responsiveness is achieved by constantly caring for it during development, or by explicitly planning for tuning measures.

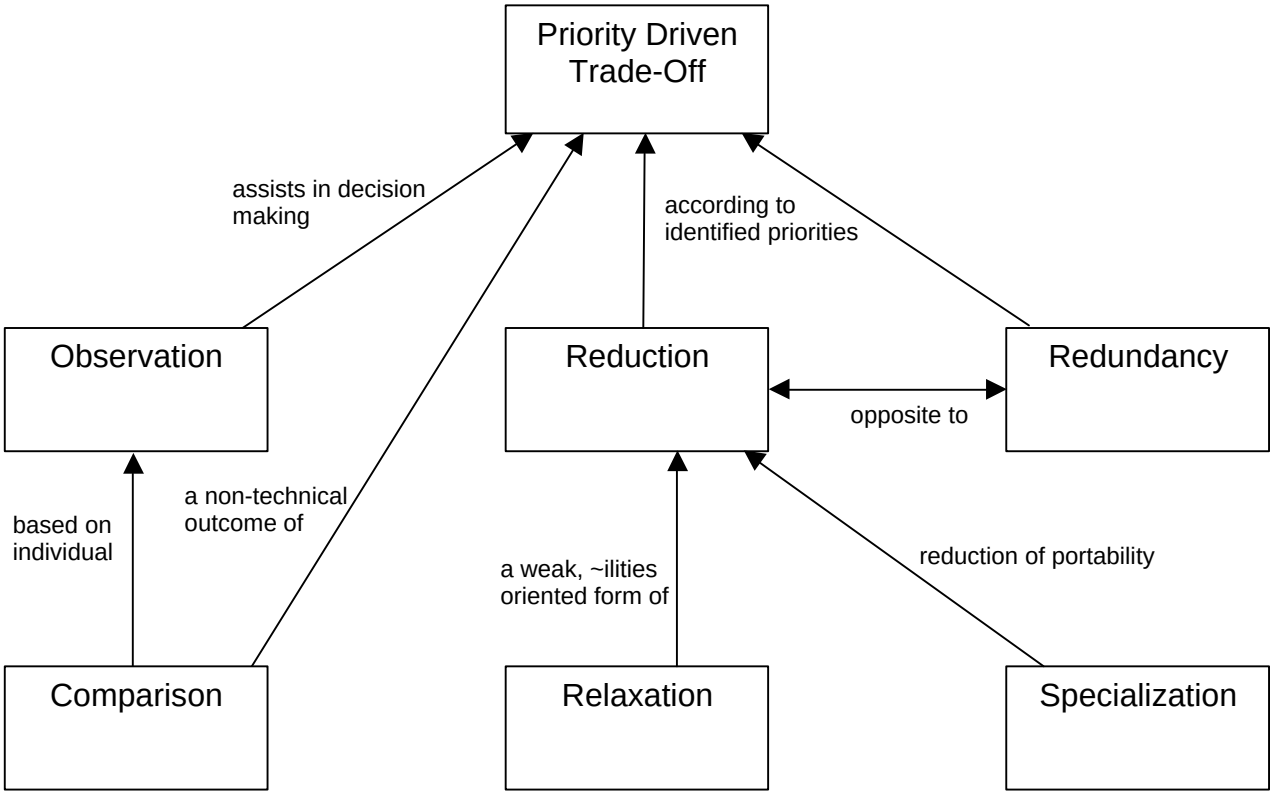
Runtime Solutions. Responsiveness is achieved by designing actions that are taken whenever responsiveness degrades during runtime, by supplying the user with information that indicates recommended actions, or surrogates a responsive system.

Most of these major strategies can be applied throughout many different domains and contexts. However, specific technical domains have found specific ways to implement the techniques; among them are Embedded Systems, Stand-alone Systems, Distributed Systems, Database Systems, and Multiprocessing Systems. These domains have large overlaps, and it is no surprise that their specific solutions are along the lines of common forces and principles.

Roadmap

The following principles and a number of their applications are presented here: PRIORITY DRIVEN TRADE-OFF, OBSERVATION, REDUCTION, REDUNDANCY, RELAXATION, SPECIALIZATION, and COMPARISON.

These principles stand in relation to each other, as the roadmap graph indicates.



PRIORITY DRIVEN TRADE-OFF is the core principle that most other principles refer to.

OBSERVATION helps to identify further measures by observing the system behavior first.

REDUCTION addresses the usage of resources that are most limited.

REDUNDANCY trades ample resources for possible slight gains in performance.

RELAXATION gives you technical and organizational hints what you might want to leave out to gain performance - and development speed.

SPECIALIZATION is a specific kind of relaxation, trading the quality of generality for performance.

COMPARISON helps you to make the performance appear good by comparing it against apparently objective criteria.

Principle: Priority Driven Trade-Off

> Users get annoyed when a cash machine takes half an eternity before it returns their card. However, they get into panic mode when it returns the card, and nothing else happens for minutes.
>
> Some graphical software may crash from time to time, but basic features are always sufficiently fast to keep the user satisfied.
>
> For a web server, you are inclined to allow for any number of incoming requests and prepare the query results. However, the user's perception is about the same whether the machine is down, or just increasingly slow because of a vicious cycle, preparing query results while all requests are interrupted due to human impatience.

You know that performance is important for your system, and you need to achieve a certain performance as a fundamental requirement.

You can trade performance against almost everything else –given you have it, and are willing to trade. This is the fundamental principle to all performance tuning. Depending on the priorities of your project, you choose a trade-off of performance against something that you are able to spend. However, be aware that most trades are neither linear nor reversible, and that each tuning measure brings its specific exchange rate.

Therefore, enumerate all the properties that are important your project and product, and prioritise them. Communicate these priorities openly, to help all participants make their own ad-hoc decisions along the lines of the big picture. For every considered tuning technique, make an explicit statement of what other qualities performance is traded against, and make your trade explicitly.

Run time performance can be improved by trading against the sum of features (scope), other quality properties of the software that have a mutual impact on performance (such as reliability, security, extensibility, almost any ~ility), hardware resources, money spend for expertise, and development time.

> *In banking systems, development effort is traded against cost of operation. E.g. a reduction of CPU load by 10% can save €50000 per month, which is worth dedicating 8 developers to performance tuning.*
>
> *The software for a patient ventilation device started with a focus towards reliability and predictable response time. After this was met, the priorities were changed to facilitate integration into a larger system, and the software interfaces were redone. Even later in the project, time to market became the major force. Because the priorities were not straight ordered but rather changed over time, the system's test facilities made sure that one achieved properties could be kept.*
>
> *A developer once felt faced with contradictory priorities in a fairly large project. So he went to the management team (a team of 5), handed them a list of about six important project and system properties, and asked to have them sorted according to their priority. The management team returned with a clear statement, that time to market, quality, and minimal cost all were number one priorities. 20 months later, that company went out of business.*

Principle: Observation

> A profiling tool allows you to determine which functions are called how often, and what processing time they consume. Still, you need to draw you own conclusions from it.
>
> Software systems may be build with reliability in mind, but the most reliable systems have an additional software or hardware layer (i.e. a watchdog) that monitors their behaviour.
>
> Your design might take great care to select effective algorithms or tune specific functions, but the user's perception of the system's performance is more likely bound on a few most frequently applied workflow scenarios.

You can improve your system's performance, you just need to know what the most appropriate measure will be.

In most systems, performance is a matter of a few critical resources that must be used with care. Over the project's course, you learn more about your application and the resources it requires. Additionally, when the functionality increases the resources you need for the application may be weighted differently than before.

Therefore, observe your performance before taking any tuning measures. Make sure you know your most severe problem before stumbling into a solution. Look out for bottlenecks, and for the most often used function. Also identify areas that are not or less constrained, and can serve you as a trading good.

A number of concrete techniques apply the principle of OBSERVATION.

Load profiles. Analyze your systems requirements and identify the major flows of work, and how frequently there are entered. Estimate the resources those scenarios consume, and create test cases on this basis. Design your system along the lines of this estimation. Where performance is critical, you might complement your analysis with a prototype.

Prototyping. For system parts in which performance is critical, create a functional prototype early in the project, and again for each major functional addition. Check what resources you need, and evaluate both the potential effect on the system, and the potential for savings that could be done in the production code.

Profiling. Analyze your current, working system and check which resources are most limited, and who consumes them. Identify a top list of resources and consumers, and focus your tuning effort there.

Budgets. Assign a performance or resource budget to individual functions or tasks. Monitor these budgets. Revise them when your prototypes or profiling indicates this. Establish the role of a resource keeper who maintains the budgets and decides on changes.

Principle: Reduction

> A real-time control system puts single values in separate network messages in order to reduce a transportation delay. Unfortunately, using ethernet the collision probablity increases and the total throughput might decrease.
>
> Components are installed by generating appropriate code from their Annotations [*Völter01*], reducing both development and startup time. However, this requires significant development effort for the middleware, and an expensive installation.

You need to achieve a certain performance as a fundamental requirement.

A lot of tactical decisions need to be taken during a project's course. You will come across situations where you can not trade against an entirely different good, but where each of your options hurts your project goals. However, not all available goods are equal, and you have your priorities and a knowledge of your system and the domain.

Therefore, reduce your usage of the most critical resource. Analyze the requirements, workflow scenarios and the existing system for the resources that you must reduce, and for resources that you could possibly trade. Make sure that you focus on usage reduction of one resource at a time, and define ranges of other goods that you a willing to spend. Additionally, analyse what actions you really need to do in which situation, and what you can do to avoid these actions at that particular moment.

A number of concrete techniques apply the principle of REDUCTION.

Reduce network traffic. Some projects are able to increase the overall throughput when they reduce the sheer number of network packets they transport. You can trade the data amount against local computation time, when you transport only compressed data. You can trade the data amount against local storage and consistency and timeliness, when you keep retrieved data in a cache. You can trade the packet number against client responsiveness or actuality, when you combine small subsequent data into packets of a size optimised for throughput. – Be aware that all these trades come with increased development costs.

Reduce data copying. Even in non-distributed systems, application developers tend to store the data in several places and to create copies when needed. While this design approach reduces the coupling between code fragments and development teams, in systems with significant data turnover the runtime costs can be diminished when the design is focused towards minimal data copying.

Reduce coupling to hardware. In embedded systems, you can sometimes dramatically decrease the coupling between your software and hardware, when you employ a powerful encapsulation layer between them such as an operating system. This trades development time against license and hardware costs. A related decision is whether you buy an encapsulation layer or build it yourself, trading development time and initial costs against license costs per unit.

Reduce run time activities. You can trade the installation and startup time against responsiveness at run time, when you use early creation of your resources, pre-loading of code and data, and code generation from annotations during installation. Alternatively, you can trade the other way round by lazy creation and online preparation.

Principle: Redundancy

> One application retrieves the same data from the server again and again, while it could know that the data is still unchanged.
>
> Another application redraws its user dialogs again and again, while it would be possible to redisplay them once they have been created.

You need to achieve a certain performance as a fundamental requirement.

Besides the major TRADE-OFFS, there are a lot of tactical decisions to take during a project's course. On a small scope, you can do a trade that is about opposite to REDUCTION – you spend more of one trading good to gain performance.

Therefore, explore what resources you have plenty of, or could have plenty of – and then go ahead and spend it. Seek for opportunities to trade this resource against something else that is less available. It is very helpful for a project when you can define this in advance, similar to your initial definition of what goals are most important to meet.

A number of concrete techniques apply the principle of REDUNDANCY.

Project redundancy. On an enterprise level it might make sense to have two projects with similar goals run in parallel, just to increase your chance for an early and huge market share. [Admittedly, I have never seen this applied intentionally, except for the fictitious example in *DeMarco97*]

Caching, data redundancy. There are uncountable implementations of caching. Most obvious are the disk caches of an operating system that trade memory against increased disk read responsiveness. Several database engines offer a client specific cache, which trades local memory against network load and server resources.

Duplication, data redundancy. Sophisticated distributed database systems support multiple server that each hold the data that their clients access most frequently. In embedded systems, the entire static memory contents is duplicated into the faster dynamic memory. CPU's load parts of frequently executed or anticipated code from main memory into their local cache. Some applications hold data in different representations at the same time, to trade for minimal access time independent of the client kind that might prefer a specific data format. This becomes especially important in combination with frequent data copying, see *Reduce data copying* above.

Duplication, code redundancy. When you abandon the ideas of code reuse and a common code base, you can gain separation between different developers or development teams and an increased development speed. Take care that the higher integration effort does not over-compensate for your savings, and beware of the long-term effect that you need to maintain multiple software systems consistently in parallel.

Pooling. You can keep instances of resources that are expensive to create, such as database or network connections, beyond their initial usage, to have them readily available the next time you need them. This trades memory, and possibly network load and scalability, against runtime responsiveness.

Principle: Relaxation

> A project defines and writes input documents, output deliverables, and status reports because the company policy demands this, but the team consists just of you.
>
> A project team defines and implements a customisable access layer to a database that will never be used again.

You need to achieve a certain performance as a fundamental requirement. You might also need to achieve a higher development performance, for which the same principle holds.

A lot of tactical decisions need to be taken during a project's course. These decisions are based on your (and others') experiences from previous projects, and most are just based on good-will assumptions or wild guessing. If somebody is not familiar with the application domain, with the development process, or with the applied technology, he may fail to see the obvious shortcut that would reduce effort and increase performance. Also, if somebody is only familiar with the particular domain, process and technology, he may fail to see the obvious shortcut another perspective would have given. Almost always you can achieve the same by making less.

Therefore, analyse what the purpose of an action in your process or application would be, and strive to replace the action by precaution. Explore what you can avoid, and relax wherever possible. Go through the project and system, look at each item and ask "can I live with less of this?" Take nothing for granted, and gain an understanding both of your project's world, and of the world outside your project.

A number of concrete techniques apply the principle of RELAXATION.

Relax on formal documentation. Most projects can increase their development speed when they reconsider their documentation effort. Each formal document needs to fulfil a real purpose, that can rarely be justified by "for the records". My favourite examples are use cases and status reports – documents where most people feel a need to fill in quite some detail to get them correct, or refuse to fill in sufficient information. Essentially, the level of detail provided does not correspond to the intention of the document and finds no audience, so that other kinds of communication could be more efficient.

Relax on actuality. Except for technical systems, most users hardly care for a sub-minute actuality. Their daily work is more complex than what they need to type into a terminal, so there are interruptions anyway. When you ensure system consistency on clearly understood checkpoints, they will accept delayed information on changes that concurrent users committed.

Relax on your control. Before you invent the fifth GUI resource editor and the 22nd database engine with specific features and abilities just for yourself, consider using a commercial product that satisfies your key requirements including performance, or consider reusing the remainders of a previous or parallel project.

Relax on your commitment. (But be aware to check for the liabilities of this not-so-ethical advice)

> *A clever project lead, you define the boundary of your own project in a way that you are not responsible for performance issues, or plan for a scapegoat. In contracts, you may leave performance unmentioned. You may also want to define a broad variety of goals that can not possibly be meet at once, giving a perfect excuse for actually meeting none of them.*

Principle: Specialization

> A system can prepare for a generic database access mechanism like ODBC, but it would be much faster if it could use the proprietary features of Objectivity (or Oracle, Versant, DB/2, or whatever).
>
> A portable framework needs to implement numerous communication concepts that most operating system provide in a unique way.

You need to achieve a certain performance as a fundamental requirement.

A lot of tactical decisions need to be taken during a project's course. These decisions take place within your current project – but how big is your project really? Are you planning for a product family, is support of multiple platforms essential, will your products need support for decades? Or is it just here and now that you need to consider? Determine your real project's scope, and re-check for your priorities. Often you have to deal with a smaller scope than you expected, or prepare for a piecemeal growth and avoid getting lost in generality of concept.

Therefore, increase your performance by selecting the most appropriate environment available, and use it in the most efficient way including unique and special features. Depending on your project's priorities, renounce from encapsulation layers, extensibility measures and feature anticipation. Especially leave aside portability when possible.

Database selection. According to the load profile of your application, you can create performance prototypes for a number of database engines. Select the engine that scales best to your needs, and focus your application to use exactly this one. Except for a few reasons, like long-term maintenance and your own convenience during the development, it is mostly not appropriate to prepare for a change of a once selected tool or environment. Check with your project's priorities.

Database tuning. Make use of the features of your specific database and infrastructure. Shed load between different servers. Use precompiled queries, appropriate device configuration and locking granularity, pages, containers, … You will find lots of specific tuning screws that can easily increase or diminish your response times by a factor of ten [*Dunham98*].

Algorithm selection. Chose those algorithms that scale with your application. When you need a multi-indexed hash table, implement it.

Staff selection. Developers are individuals and show individual skills and behaviour. Select people that will fit into your team – check for Self Selecting Team [*Coplien95*] to avoid unnecessary friction. Of socially compatible candidates, select those that either have the specific experience that you need, or that have the ability to learn fast.

Principle: Comparison

> The new operating system provides more features than the former version, but performance degraded so that your favourite game runs less smoothly. The project took longer than you initially expected, however the team works more efficient than other teams in the company.

You know that performance is important for your system, and you need to achieve a certain performance as a fundamental requirement.

In most systems, performance is a matter of perception rather than of exact measurement. You expect a system behaviour compared to some measure that may be a weak, personal one. Most users are fine when their system is faster than they initially thought it would be, or at least faster than their old one, or better yet than the neighbors' one.

Therefore, compare your performance to some measures that make your performance look good. Allow system user to do their own comparison, and provide a reference that serves both you and the user.

Performance indicator. Though it may seem ancient history, it is actually not so long ago that in a computer retail store all monitors were busy measuring the CPU and graphics performance - compared to the original PC AT or even XT. The then best performing systems were several orders of magnitude faster than their scale, which made for very impressive performance graphs in marketing.

Illusion of feedback. Give the user feedback that the system is busy, but do not give precise estimates about the actual needed duration. A trivial implementation is the hour-glass cursor in windows based systems, or an animated emblem in a web browser. Only slightly more precise is the line of dots during system startup. More feedback is provided by progress bars, but they do not necessarily relate to the required time remaining.

Make the user worry. [*Noble+98*] Make your performance data available to the user, and let him decide whether to degrade the system functionality for the sake of higher responsiveness, or vice versa. Make commands interruptible that could take significant time, and let the user decide whether he accepts to wait for completion.

Trampoline, AKA short interrupts. [*Marquardt01*] For technical clients, react on their requests seemingly immediately. Defer the actual work to a later point in time where you are less restricted.

Conclusion

The principles of performance tuning offer a sorting criterion for numerous existing and emerging patterns related to performance issues. My hope is that by making the principles available, present literature can find a place within them, and further patterns can emerge by simple analogies. It could help people from very different backgrounds to share their experience and find a common language.

Acknowledgement

Thanks to the participants of the "Performance Patterns Language" design fest at EuroPLoP 2001. Without them, I probably would not have spent the time to collect these principles. Thanks also to James Noble and Charles Weir. Their memory preservation society on EuroPLoP 1998 brought the idea to me that a similar effort could be worthwhile for performance as well.

Special thanks are reserved for my shepherd, Jens Coldewey, for reviewing many drafts and providing examples and hints. Thanks also to the workshop participants at EuroPLoP 2002.

References

Coplien95 James Coplien: A Generative Development-Process Pattern Language. In: Pattern Languages of Program Design, Addison-Wesley 1995

DeMarco97 Tom DeMarco: The Deadline. A Novel About Project Management. Dorset House 1997

Dunham98 Jeff Dunham: Database Performance Handbook. McGraw-Hill 1998

Marquardt01 Klaus Marquardt: Performance Pattern Language. Results of the EuroPLoP 2001 Design Fest. In: Proceedings of EuroPLoP 2001

Noble+98 James Noble, Charles Weir: Proceedings of the Memory Preservation Society. In: Proceedings of EuroPLoP 1998

Noble+00 James Noble, Charles Weir, Duane Bibby: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley 2000

Völter01 Markus Völter: Server Side Components – A Pattern Language. In: Proceedings of EuroPLoP 2001